MONASH University

ECE3073 **Computer Systems**

Program Design and Analysis: Assembling and Linking

ECE3073/TRC3300

Acknowledgement

Based on

The lecture notes of Marilyn Wolf

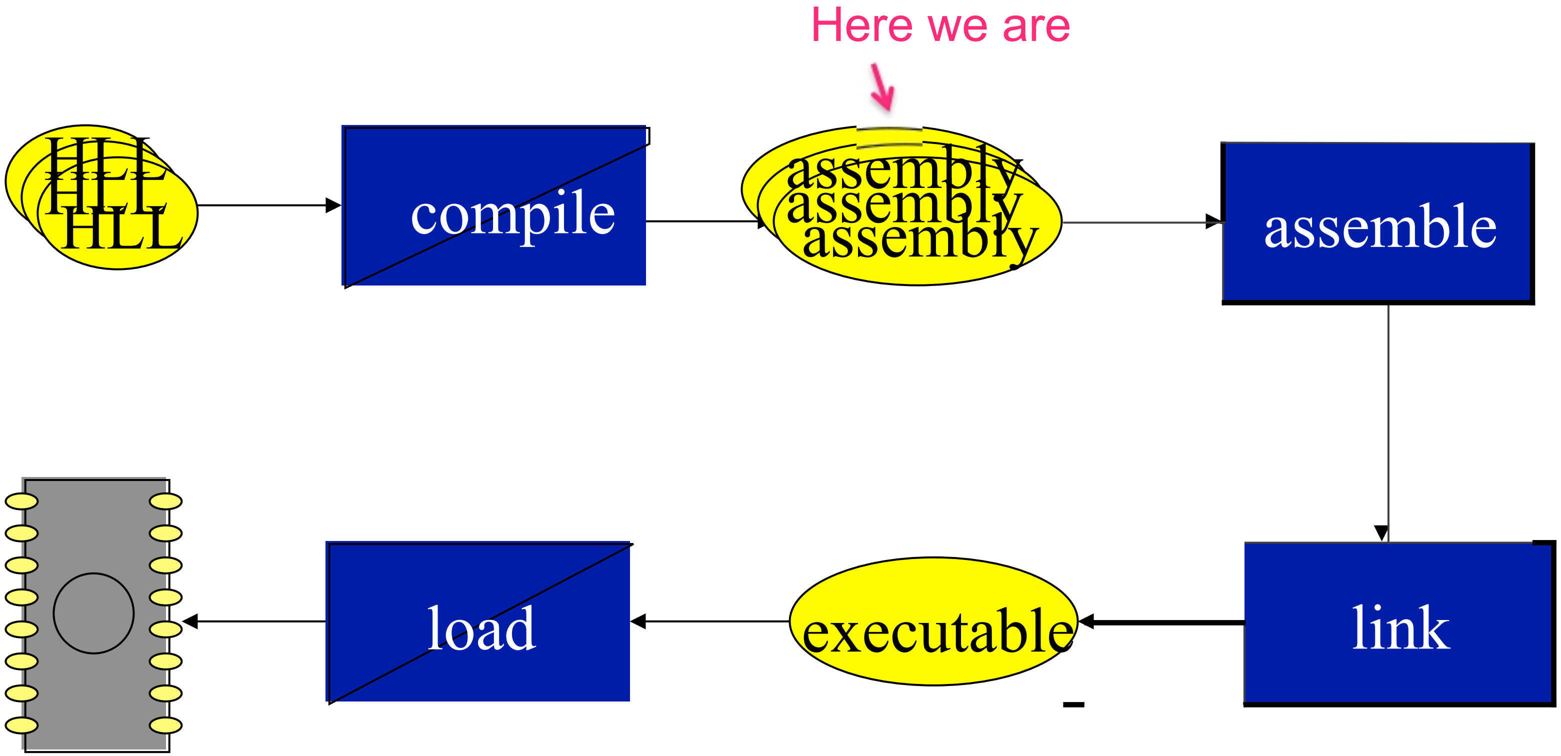
Computers as Components, Principles of Embedded Computing System Design

And adaptations from Dr Royan Ong, Malaysian Campus

Minor modifications Clive Maynard 2020

WARNING

C to Binary


Here we are
HLL
HLL
HLL
compile
assembly
assembly
assembly
assemble
link
executable
load

Multiple-module programs

- **Programs may be composed from several source files**
- **Addresses become more specific during processing:**
  - relative addresses: measured relative to the start of a module
  - absolute addresses: measured relative to the start of the CPU address space.

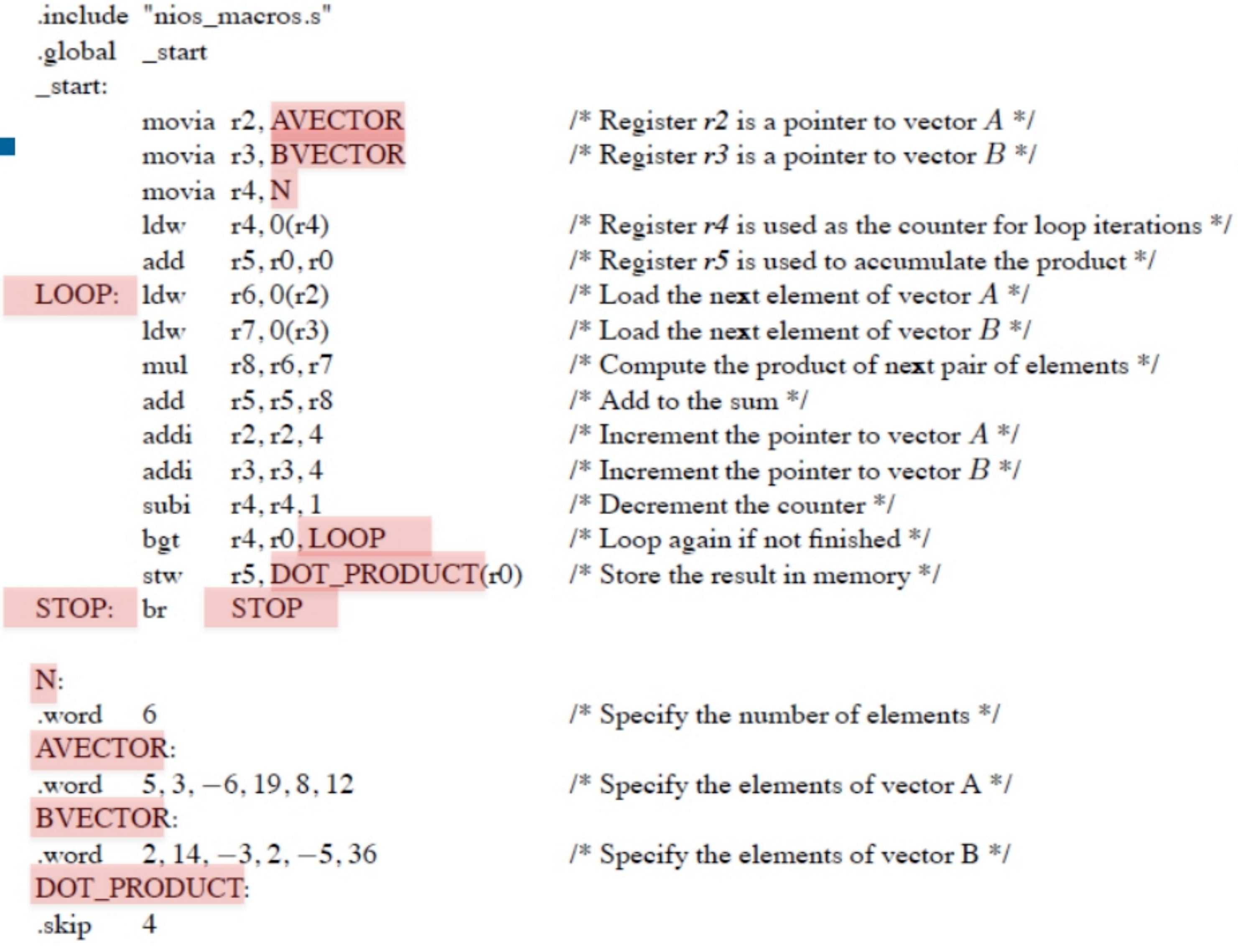
Assemblers

- **Major tasks:**
  - translate labels into addresses
  - handle pseudo-ops (data, etc.)
  - generate binary for symbolic instructions (mnemonics)
  - Generally one-to-one translation
- **Assembly labels**

```
        .ORG 100
Label1:  ADD  r1, r2, r3
```

Note: Many RISC processors have fixed size instructions which makes the memory allocation process for instructions much easier than with the CISC variable length instruction requirements.

Nios Introduction



```
.include  "nios_macros.s"
.global   _start
_start:
          movia  r2, AVECTOR               /* Register r2 is a pointer to vector A */
          movia  r3, BVECTOR               /* Register r3 is a pointer to vector B */
          movia  r4, N
          ldw    r4, 0(r4)                 /* Register r4 is used as the counter for loop iterations */
          add    r5, r0, r0                /* Register r5 is used to accumulate the product */
 LOOP:    ldw    r6, 0(r2)                 /* Load the next element of vector A */
          ldw    r7, 0(r3)                 /* Load the next element of vector B */
          mul    r8, r6, r7                /* Compute the product of next pair of elements */
          add    r5, r5, r8                /* Add to the sum */
          addi   r2, r2, 4                 /* Increment the pointer to vector A */
          addi   r3, r3, 4                 /* Increment the pointer to vector B */
          subi   r4, r4, 1                 /* Decrement the counter */
          bgt    r4, r0, LOOP              /* Loop again if not finished */
          stw    r5, DOT_PRODUCT(r0)       /* Store the result in memory */
 STOP:    br     STOP

N:
.word     6                                /* Specify the number of elements */
AVECTOR:
.word     5, 3, −6, 19, 8, 12              /* Specify the elements of vector A */
BVECTOR:
.word     2, 14, −3, 2, −5, 36             /* Specify the elements of vector B */
DOT_PRODUCT:
.skip     4
```

Figure 6. A program that computes the dot product of two vectors.

Two-pass assembly

- **Pass 1:**
  - Generate Symbol Table
- **Pass 2:**
  - Generate binary instructions (object module, often *.o extension)

Symbol Table

- **Table used by compiler and assembler where each identifier from the source code is associated with information relating to its declaration such as type and location**

Program Location Counter

Assembly Code

```
0            ADD r0,r1,r2
4    L1:     ADD r3,r4,r5
8            CMP r0,r3
C    L2:     SUB r5,r6,r7
```

Symbol Table

L1	0x00000004
L2	0x0000000C

Symbol Table Generation

- Program location counter (PLC) holds address location of each instruction
- Assembler scans assembly program and increments PLC as it goes along
- When labels encountered, placed in Symbol Table together with PLC value
- the place in the file where a label is defined is known as the **entry point**
- The place where a label is used is called an **external reference**

Relative Address Generation

- **Some label values are unknown at assembly time**
- **Label values within each module kept in relative (to the start of the module) form**
- **Must keep track of external labels, cannot generate binary (executable program) for instructions that use external labels**
  - Example: code in Module A calling code in Module B, Module A has an external label.

Pseudo-operations

- **Pseudo-ops do not generate instructions:**
  - ORG: sets instruction starting location
  - EQU: generates symbol table entry without advancing PLC
  - Data statements define data blocks

Example (1/5): Source codes

- **Source codes in two C files for PIC18F8722 microcontroller**

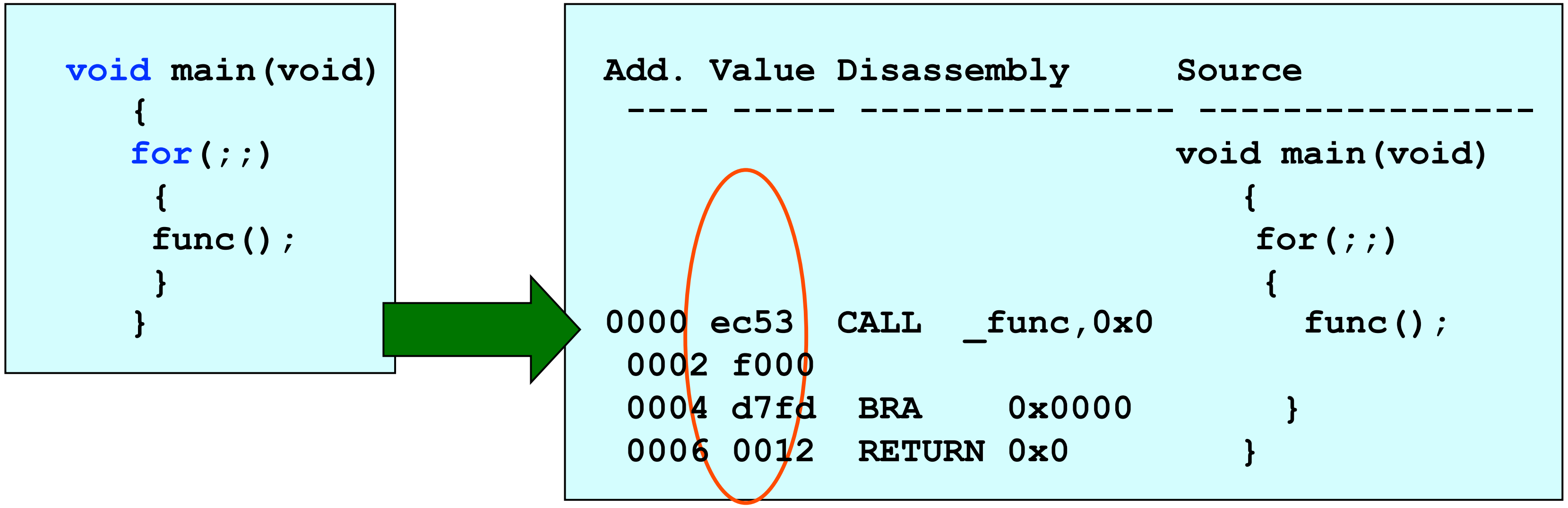
```
void main(void)
 {
 for(;;)
     {
     func();
     }
 }
```

```
void func(void)
 {
 unsigned char i, count;

 for(i = 0; i < 100; i++)
     {
     count++;
     }
 }
```

Example (2/5): Compilation and Assembly

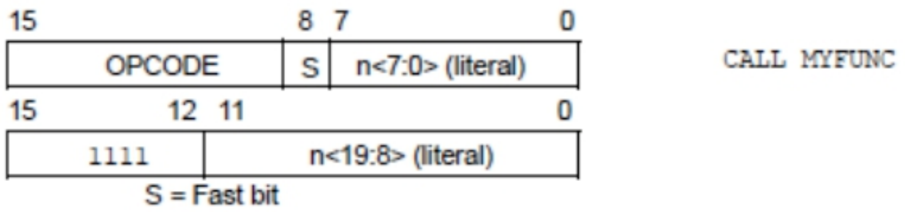
- **main.c to assembly to machine code**



```
void main(void)
   {
   for(;;)
    {
    func();
    }
   }
```

```
Add. Value Disassembly     Source
 ---- ----- --------------- ----------------
                           void main(void)
                              {
                               for(;;)
                                {
0000 ec53  CALL  _func,0x0        func();
 0002 f000
 0004 d7fd  BRA    0x0000         }
 0006 0012  RETURN 0x0          }
```

- **Machine code produced**
- **Symbol Table needs to hold _func (address cannot be resolved at this stage)**



Example (3/5): Compilation and Assembly

- **func.c to assembly to machine code**

```
void func(void)
 {
 unsigned char i, count;

   for(i = 0; i < 100; i++)
    {
    count++;
    }
   }
```

```
Add. Value Disassembly         Source
---- ----- -------------------- ----------------------------
0000 cfd9  MOVFF  0xfd9,0xfe6   void func(void)
0002 ffe6
0004 cfe1  MOVFF  0xfe1,0xfd9
0006 ffd9
0008 0e02  MOVLW  0x2
000A 26e1  ADDWF  0xe1,0x1,0x0
                                   {
                                   unsigned char i, count;
000C 6adf  CLRF  0xdf,0x0          for(i = 0; i < 100; i++)
000E 0e64  MOVLW  0x64
0010 5cdf  SUBWF  0xdf,0x0,0x0
0012 e204  BC    0x001c
0018 2adf  INCF  0xdf,0x1,0x0
001A d7f9  BRA   0x000e
                                      {
0014 0e01  MOVLW  0x1                count++;
0016 2adb  INCF  0xdb,0x1,0x0
                                      }
001C 0e02  MOVLW  0x2              }
001E 5ce1  SUBWF  0xe1,0x0,0x0
0020 e202  BC    0x0026
0022 6ae1  CLRF  0xe1,0x0
0024 52e5  MOVF  0xe5,0x1,0x0
0026 6ee1  MOVWF  0xe1,0x0
0028 52e5  MOVF  0xe5,0x1,0x0
002A cfe7  MOVFF  0xfe7,0xfd9
002C ffd9
002E 0012  RETURN 0x0
```

- **Relative address**

MONASH University

Example (4/5): Symbol Table

- **Partial symbol table listing**

```
         Name   Address   Location     Storage File
   ---------- ---------  ----------  ---------- ---------
       __init  0x000108    program      extern C:\MCC18\src\traditional\stdclib\__init.c
 __zero_memory 0x0000f2    program      extern C:\MCC18\src\traditional\proc\p18F8722.asm
    _do_cinit  0x000008    program      extern C:\MCC18\src\traditional\startup\c018i.c
       _entry  0x000000    program      extern C:\MCC18\src\traditional\startup\c018i.c
     _startup  0x0000d6    program      extern C:\MCC18\src\traditional\startup\c018i.c
         func  0x0000a6    program      extern G:\temp\test\func.c
         main  0x000100    program      extern G:\temp\test\main.c
```

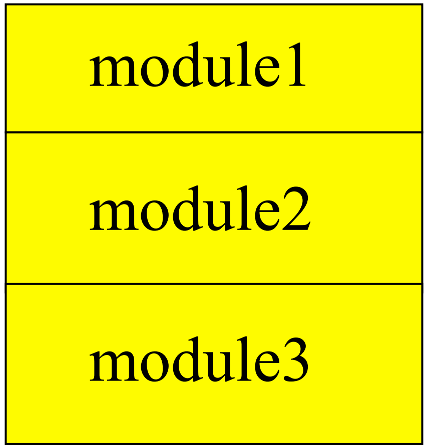
- **Additional preassembled code segments (e.g. _startup) linked by the linker to produce executable**

Linking

- **Combines several object modules into a single executable module**
- **Jobs:**
  - Order modules
  - Resolve labels across modules (external labels)
  - Generate single executable

Module ordering

- Object code modules must be placed in absolute positions in the memory space
- Load map or linker flags control the order of modules



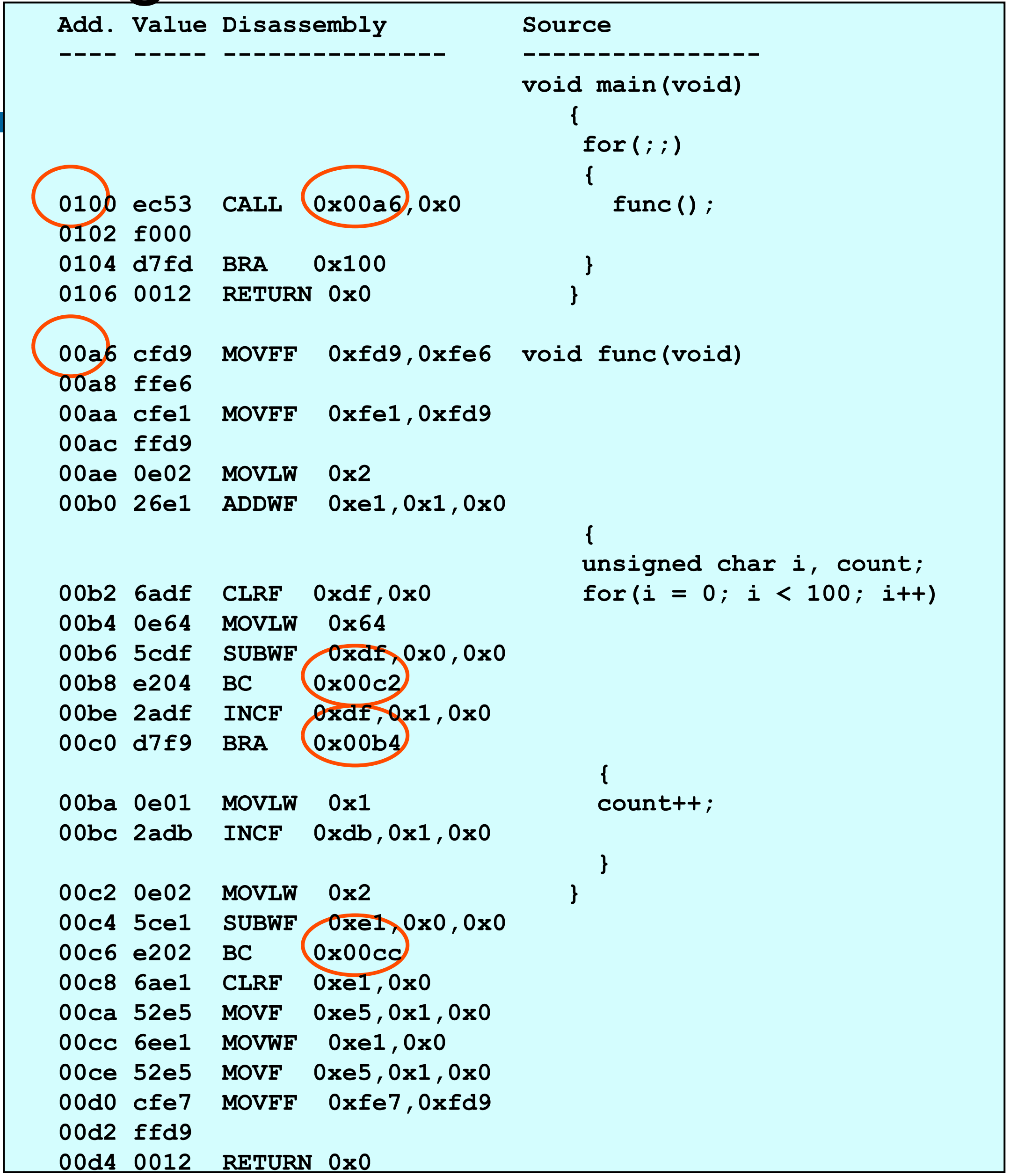
module1

module2

module3

Example (5/5): Linking

- **Final code:**
  - main.c relocated to 0x0100
  - func.c relocated to 0x00A6
  - Changes from relative to absolute addresses
  - Additional startup code added by compiler not shown, starts from 0x0000



```
Add. Value Disassembly          Source
---- ----- ----------------     ----------------
                               void main(void)
                                  {
                                   for(;;)
                                   {
0100 ec53  CALL  0x00a6,0x0          func();
0102 f000
0104 d7fd  BRA   0x100             }
0106 0012  RETURN 0x0             }

00a6 cfd9  MOVFF  0xfd9,0xfe6  void func(void)
00a8 ffe6
00aa cfe1  MOVFF  0xfe1,0xfd9
00ac ffd9
00ae 0e02  MOVLW  0x2
00b0 26e1  ADDWF  0xe1,0x1,0x0
                                   {
                                   unsigned char i, count;
00b2 6adf  CLRF  0xdf,0x0          for(i = 0; i < 100; i++)
00b4 0e64  MOVLW  0x64
00b6 5cdf  SUBWF  0xdf,0x0,0x0
00b8 e204  BC    0x00c2
00be 2adf  INCF  0xdf,0x1,0x0
00c0 d7f9  BRA   0x00b4
                                    {
00ba 0e01  MOVLW  0x1               count++;
00bc 2adb  INCF  0xdb,0x1,0x0
                                    }
00c2 0e02  MOVLW  0x2             }
00c4 5ce1  SUBWF  0xe1,0x0,0x0
00c6 e202  BC    0x00cc
00c8 6ae1  CLRF  0xe1,0x0
00ca 52e5  MOVF  0xe5,0x1,0x0
00cc 6ee1  MOVWF  0xe1,0x0
00ce 52e5  MOVF  0xe5,0x1,0x0
00d0 cfe7  MOVFF  0xfe7,0xfd9
00d2 ffd9
00d4 0012  RETURN 0x0
```

Dynamic linking

- **Some operating systems link modules dynamically at run time:**
  - shares one copy of library among all executing programs
  - allows programs to be updated with new versions of libraries

- dll